

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy**, as well as for **helpful advice** for how to write up your solutions.

1. (4 pts) Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

[We are expecting: a detailed proof, using the definition of a binary search tree.]

SOLUTION.

Without loss of generality, consider the case where x is the left child of y . We will show that $y.key$ is the smallest key in T that is larger than $x.key$. If x was the right child of y , we would analogously show that $y.key$ was the largest key in T that was smaller than x .

Consider any node r whose key is greater than x . We show that either r equals y , or $r.key > y.key$.

Any pair of nodes in the tree must have a lowest common ancestor, so in particular, r and x must have a lowest common ancestor, call it s . Since x is a leaf node, s cannot be x . So either $s = r$, or s is neither x nor r .

Case 1: s is not equal to r . If s is neither x nor r , then x must be in the left subtree of s , and r must be in the right subtree of s . (This is because if they were both in the same subtree of s , then they would have a common ancestor that was lower than s , and if x was in the right subtree of s and r was in the left subtree, that would violate our assumption that r 's key was greater than x .) Now either y equals s (in which case $r.key > y.key$ by the binary search tree property), or s is an ancestor of y . If s is an ancestor of y , then y must also be in the left subtree of s , because otherwise y could not be the parent of x . By the binary search tree property, $y.key < s.key < r.key$.

Case 2: s equals r . In this case, r must be an ancestor of x . So either r equals y , in which case we are done, or r is an ancestor of y . Now there are two cases if r is an ancestor of y . Either y is in the left subtree of r , or y is in the right subtree of r . If y is in the left subtree of r , then $y.key < r.key$. If y is in the right subtree of r , then x is also in the right subtree of r , contradicting the statement that $r.key > x.key$. This concludes the proof.

2. **Open-addressing.** Read CLRS Section 11.4, pages 269-276, on open-addressing. The answers to the following questions may be implicit in the text: please answer them in your own words. Suppose that in the following, we consider an open-addressed hash table with n buckets, and a universe of size $M \geq n(n+1)$.

- (a) (1 pt) Suppose that we never delete items from an open-addressed hash table. After inserting $m \leq n$ items, what is the *worst-case*¹ time to **search**?

¹As always, this means that an adversary who knows everything about the data structure gets to choose m items to insert, and then chooses an item to search for.

[We are expecting: your bound, and an informal argument that it is correct; this includes both an argument that your bound could happen in the worst case, and an argument that the search time won't be any worse than that. A paragraph should be enough.]

- (b) (4 pts) In the discussion about deletions on page 271, we store the value DELETED instead of NIL when an item is deleted. Suppose we did not do this, and simply deleted items by returning their value to NIL. Suppose, as above, that there are currently $m \leq n$ items in the table, but that any number may have been inserted and deleted before. Give a correct algorithm to search for an element, if deletions of the form above were allowed. What is the worst-case time for this search algorithm to find an element? Above, "correct" means that $\text{search}(x)$ will always return x if it is the table (and will never return anything if x is not in the table). Argue that any correct algorithm must have worst-case running time at least this.

[We are expecting: a description of a correct search algorithm, and informal arguments about its worst-case running time and about why any other correct search algorithm must have worst-case running time at least this. A paragraph or two should be enough.]

- (c) (3 pts) In this question, you will be adversarially generating input for a open-addressed hash table. You insert (in order) the keys $1, 2, 3, \dots, m$. Given that the next operation will be **search**, you get to choose a key in $\{1, \dots, m\}$ to search for that will *maximize* the time that the operation takes. You do not know the probe sequence, so there is no way to know the exact answer to this question. Nonetheless, what key would you choose and why?

[We are expecting: an answer and some informal reasoning (a few sentences) about why you'd hope this would maximize the search time.]

SOLUTION.

- (a) The worst-case time is $O(m)$. Nothing can take longer than this, because the longest probe sequence cannot be longer than m if there are only m items in the table.

We also should show that this time is worst-case by explaining what that worst case is. For this problem, it was okay to consider any particular probing scheme discussed in CLRS. The first solution below applies to linear probing and quadratic probing. For double-hashing and uniformly random probing, see the second argument below.

With linear hashing and quadratic hashing, two elements that hash to the same initial bucket have the same probe sequence. By the pigeonhole principle, because $M \geq n(n+1)$, there are at least $n+1 \geq m+1$ items that hash to the same initial bucket. So the adversary would insert m of these $m+1$ items, and then search for the $m+1$ 'st of them.

For other probing schemes discussed in CLRS, notice that as long as M is sufficiently larger than n , these schemes have the following property: there is some x so that for any cell i of the hash table, there is some element $y_i \neq x$ so that the first

probe for y_i is i . Further, the first $m \leq n$ probe locations for x (that is, $h(k, x)$ for $k = 1, \dots, m$) are distinct.

Using this property, let's see how the adversary would choose to hash things. First, he will choose an item, x , to eventually search for. Suppose that $h(k, x) = i_k$, and all the i_k are distinct. Then the adversary will insert $y_{i_1}, y_{i_2}, \dots, y_{i_m}$. Now, when the adversary chooses to search for x , it will take time $\Omega(m)$. This is because he will first look at bucket $h(1, x)$, and find it occupied by y_{i_1} . Next he will look at bucket $h(2, x)$, find it occupied by y_{i_2} , and so on, until he looks at bucket $h(m, x)$. This takes m steps.

- (b) A correct algorithm would have to search the entire probe sequence for a key, which might be size $O(n)$; so this is the worst-case search time. (We would also accept infinite time, depending on how you described the search algorithm). The reason that we have to search the whole table is as follows.

As in part (a), we will first give an answer for the simpler case of linear or quadratic probing (which was fine for this problem). Because $M \geq n(n+1)$, there are $n+1$ items that hash initially to the same bucket; hence, for linear or quadratic probing, they have the same probe sequences. Suppose these items are $x_0, \dots, x_n$, and we are going to search for x_0 . Then any of the following things are possible. First, for any $k = 1, \dots, n$, it's possible that x_0 could be at the location $h(k, x_0)$, while all the other locations $h(j, x_0)$ for $j \neq k$ are NIL values. For example, this could happen if $x_1, \dots, x_{k-1}$ were inserted, then x_0 was inserted, then $x_1, \dots, x_{k-1}$ were deleted without markers. Second, it's possible that $h(k, x_0)$ are NIL for all k , if nothing was ever inserted. In order to distinguish between these possibilities, we must look at the entire probe sequence $h(k, x_0)$ for $k = 1, \dots, n$. So this takes time $\Omega(n)$.

That reasoning works for linear or quadratic probing. As in part (a), we can make a similar argument using the property that for some x (which we are going to search for), for every i there is some $y_i \neq x$ so that $h(1, y_i) = i$. Again, if the universe is large enough this is true for any reasonable probing sequence: it says that if you remove one element (x) from the universe, the remaining ones will still cover the hash table with their first probes.

Now, as in the first argument, there are many different things we need to distinguish between. First, fix x and suppose the probe sequence is $i_1, \dots, i_n$. Then consider the following scenarios. First, if $y_{i_1}, \dots, y_{i_{k-1}}$ were inserted, then x were inserted, then $y_{i_1}, \dots, y_{i_{k-1}}$ were deleted, then we'd be left with everything NIL except for x at i_k . Additionally, maybe the whole table is empty. To be able to distinguish between these we need to look at the entire probe sequence.

- (c) The worst thing to do would be to search the key m again. Without any other information, this is the key that is most likely to have gotten bumped a lot, and so it's the key that will likely require a lot of probing to find.
3. In this problem, we will investigate a way to improve the worst-case performance of the chained hash tables that we saw in class. Suppose that $\mathcal{U}$ is a universe of size M , let n be an integer and that $\mathcal{H}$ is a family of hash functions that map $\mathcal{U}$ into buckets labeled

$1, \dots, n$. Suppose that $M \geq n^2$. Suppose that $\mathcal{H}$ is a universal hash family.

- (a) (2 pt) What is the worst-case running time for **search** in a chained hash table? That is, suppose that after h is chosen, an adversary chooses $u_1, \dots, u_n \in \mathcal{U}$ to insert into the table, and then runs a **search** query on an item of their choosing; how long will this last **search** query take in the worst case? **[We are expecting: One or two sentences with your answer, and a description of how this might happen.]**
- (b) (4 pts) Find a way to modify the chained hash table that we saw in class so that, if at most n items $u_1, \dots, u_n \in \mathcal{U}$ are ever inserted into your modified data structure:
 - The expected time² to perform **search, insert, delete** is still $O(1)$.
 - The worst-case running time (in the sense of part (a)) to perform **search, insert, delete** operations are $O(\log(n))$.

Your modified data structure should still involve choosing a hash function $h \in \mathcal{H}$ uniformly at random, and should still use only $O(n \log(M))$ space. **[We are expecting: a description of the data structure and a description of how to perform search, insert, and delete in this data structure. We are also expecting an informal argument (a paragraph or two) about the expected and worse case running time. You do not need to write a formal proof, and you may appeal to results and arguments from class or the textbook.]**

SOLUTION:

- (a) The worst-case time is $\Omega(n)$. This could happen as follows. Suppose that $h \in \mathcal{H}$ is drawn. By the pigeonhole principle, because $M \geq n^2$, there are items $x_1, \dots, x_n$ that all hash to the same bucket under h : $h(x_1) = \dots = h(x_n)$. The adversary would insert $x_1, \dots, x_n$, and then search for $x_{n/2}$. Then the linked list in bucket $h(x_1)$ has n things in it, so searching for one of them in the middle will take time $\Omega(n)$. (It is also find to search for x_1 or x_n : at least one of them will take time $\Omega(n)$, depending on how the linked list is implemented).
- (b) We will replace the linked lists in the description of a chained hash table with red-black trees. Formally, let A be an array of length n , and instantiate each entry of A with a pointer to an empty red-black tree. Choose a random hash function $h \in \mathcal{H}$. Then to do any operation $\text{op} \in \{ \text{search}, \text{insert}, \text{delete} \}$, on an item x , we perform $\text{op}(x)$ on the red-black tree rooted at $A[h(x)]$.
 We first notice that the expected time for any operation does not change: just as in class (or in CLRS), for all u_i , the expected number of u_j so that $h(u_i) = h(u_j)$ is $O(1)$. This means that the expected size of the red-black tree rooted at $A[h(u_i)]$ is $O(1)$, and any operation on it takes time $O(1)$ in expectation.
 Next we observe that the worst-case time is now $O(\log(n))$. Even if all n items land in the same bucket, the guarantee of the red-black tree is that the worst-case time to do any operation is $O(\log(n))$.

²This is expected time in the formal sense discussed in class: for any fixed set of items $u_1, \dots, u_n \in \mathcal{U}$, and for any sequence of L **search, insert, delete** operations only involving these elements, the expected amount of time, over the randomness used to choose the data structure, to perform all L operations is $O(L)$.

Finally, we observe that the size of the hash table is still small, since it is exactly the same as the original data structure. Instead of the linked lists, we are now storing the same amount of data in tree form.

Grading note: while we have addressed the size above, this is not required for credit. (Since we did not dwell much on the exact size bound of the data structure in class, just on the size of the hash family).

4. In this problem, we'll investigate the definition of universal hash functions. Let $\mathcal{U}$ denote a universe of size M , and let n be the number of buckets in a hash table.

- (a) (3pts) Let $\mathcal{H}$ be the family of all possible functions mapping from $\mathcal{U}$ to $\{1, \dots, n\}$. Prove that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h randomly chosen from $\mathcal{H}$,

$$\Pr[h(x_i) = h(x_j)] = \frac{1}{n}.$$

This shows that $\mathcal{H}$ is a universal hash family, and moreover that we have *equality* in the definition of the universal hash family property, not just a $\leq$ relationship.

[We are expecting: a careful and rigorous proof, though it should not need to be more than one or two paragraphs. You should be especially careful about what is random and what is not.]

- (b) (3pts) There also exist hash families such that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h randomly drawn from the family, $\Pr[h(x_i) = h(x_j)] < \frac{1}{n}$. (Notice that the inequality here is strict!) We will now explore one such family. Consider $\mathcal{H}' = \mathcal{H} \setminus \{h_1\}$, where h_1 is the function defined by

$$h_1(x_i) = 1 \quad \text{for all } x_i \in \mathcal{U}.$$

That is, $\mathcal{H}'$ is the family of all functions from $\mathcal{U}$ to $\{1, \dots, n\}$ *except* for the function h_1 which sends all elements of $\mathcal{U}$ to 1.

Prove that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h drawn randomly from $\mathcal{H}'$, we have

$$\Pr[h(x_i) = h(x_j)] < \frac{1}{n}.$$

[We are expecting: a clear and rigorous proof. As above, this needn't be more than a paragraph of two, but you should be very careful about what is random and what is not.]

- (c) (0pts) **[BONUS: Part (c) is NOT REQUIRED but might be fun to think about. It will not affect your grade but we will give you feedback if you answer it.]** Give a hash family $\mathcal{H}$ so that

$$\max_{x \neq y \in \mathcal{U}} \Pr[h(x) = h(y)]$$

is as small as possible (and ideally prove that it is as small as possible). What is this probability? Above, the probability is over the choice of a uniformly random $h \in \mathcal{H}$.

[We are expecting: we aren't expecting anything, this is a bonus problem.]

Solution:

- (a) We give several solutions below.

Solution 1. We can break up the probability into the sum of conditional probabilities based on the value of $h(x_i)$. That is, $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^n \Pr[h(x_i) = k] \cdot \Pr[h(x_i) = h(x_j) | h(x_i) = k]$. But what is $\Pr[h(x_i) = h(x_j) | h(x_i) = k]$ exactly? It is in fact just $\Pr[h(x_j) = k]$, since once we know that $h(x_i) = k$, the only way for $h(x_j)$ to equal $h(x_i)$ is if $h(x_j) = k$. Then $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^n \Pr[h(x_i) = k] \cdot \Pr[h(x_j) = k]$.

Now, because $\mathcal{H}$ consists of every possible function from $\mathcal{U}$ to $\{1, \dots, n\}$, we know that there are equally many functions mapping x_i to every possible value in $\{1, \dots, n\}$. Then $\Pr[h(x_i) = k]$ is exactly $1/n$, and similarly for $\Pr[h(x_j) = k]$. This tells us that $\Pr[h(x_i) = h(x_j)] = \sum_{k=1}^n \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n}$, since we are summing $(\frac{1}{n})^2$ up n times. This proves that indeed the probability of collision is indeed exactly $1/n$, as desired.

Solution 2: The number of hash functions so that $h(x_i) = h(x_j) = k$ is n^{M-2} for each $k = 1, \dots, n$, because there are $M - 2$ remaining elements of $\mathcal{U}$ that can be assigned. So the total number of hash functions so that $h(x_i) = h(x_j)$ is exactly n^{M-1} . Thus the probability that $h(x_i) = h(x_j)$ is exactly

$$\frac{n^{M-1}}{|\mathcal{H}|} = \frac{n^{M-1}}{n^M} = \frac{1}{n}.$$

Note: We will also accept correct solutions that start from the fact that a uniformly random function $h : \mathcal{U} \rightarrow \{1, \dots, n\}$ has that $\{h(x_i) \mid x_i \in \mathcal{U}\}$ is a collection M of i.i.d. uniformly random variables in $\{1, \dots, n\}$.

- (b) One way to see that this is the case is to notice that, by removing h_1 from the set of possible functions, we are making sure that every pair of elements is strictly less likely to collide. That is, in part (a), the probability we got was based on the existence of this function in the set, and removing the function must make the probability of collision strictly lower.

We can also (and should, to get full credit!) prove this more formally. Fix $x_i, x_j \in \mathcal{U}$. We will count the number of $h \in \mathcal{H}$ that have $h(x_i) = h(x_j)$.

First, observe that for all $k \in \{2, \dots, n\}$, the number of h so that $h(x_i) = h(x_j) = k$ is exactly n^{M-2} . That is, for each of the remaining $M - 2$ items of $\mathcal{U}$, we have n choices each. The fact that $\mathcal{H}'$ is missing h_1 does not affect this computation, since $h(x_i) = k \neq 1$, so $h \neq h_1$ in this case.

On the other hand, for $k = 1$, the number of h so that $h(x_i) = h(x_j) = 1$ is exactly $n^{M-2} - 1$. The reasoning is exactly the same as before, except we throw h_1 out of the count.

Then, the probability

$$\begin{aligned}
\Pr h(x_i) = h(x_j) &= \frac{\text{number of } h \in \mathcal{H}' \text{ so that } h(x_i) = h(x_j)}{|\mathcal{H}'|} \\
&= \frac{n^{M-2} - 1}{n^M - 1} + \sum_{k \neq 1} \frac{n^{M-2}}{n^M - 1} \\
&= \frac{n^{M-1} - 1}{n^M - 1} \\
&< \frac{1}{n}.
\end{aligned}$$

- (c) (Bonus problem). [Note: this solution is sketchier than normal because the problem is a bonus problem. Please ask if you want more details on any step!!] Let H be any hash family, and draw h from H . Suppose that $n|M$ for simplicity. First, we compute

$$\begin{aligned}
\sum_{x,y \in \mathcal{U}} \Pr(h(x) = h(y)) &= E \sum_{x,y \in \mathcal{U}} \mathbf{1}(h(x) = h(y)) \\
&= E \sum_{j=1}^n |B_{j,h}|^2
\end{aligned}$$

where

$$B_{j,h} = \{x \in \mathcal{U} \mid h(x) = j\},$$

and where the probability and expectation are both over h . Breaking up the first sum into pairs x, y where $x = y$ and pairs where they are not equal, we have

$$M + \sum_{x \neq y} \Pr(h(x) = h(y)) = E \sum_{j=1}^n |B_{j,h}|^2,$$

hence

$$\max_{x \neq y} \Pr(h(x) = h(y)) \geq \frac{E \left[\sum_{j=1}^n |B_{j,h}|^2 \right] - M}{M(M-1)}.$$

Now, the sum in the right hand side is minimized when all of the buckets $B_{j,h}$ are the same size; that is, size M/n . (This follows from Cauchy-Schwartz). Plugging this in, we compute

$$\max_{x \neq y} \Pr(h(x) = h(y)) \geq \frac{M/n - 1}{M - 1}.$$

The reasoning above also shows that this is tight: if we choose $\mathcal{H}$ to be the set of functions that are “balanced” (that is, $\mathcal{H}$ is the set of all functions $h : U \rightarrow \{1, \dots, n\}$ so that $|B_{j,h}| = M/n$ for all j), then all of the inequalities above are actually equalities.

5. Suppose that you are making a lightweight web browser and you have a large, fixed blacklist of w malicious websites. Let M denote the number of websites on the whole internet (this is a huge number!) and suppose for this problem that this number does not change.

Your goal is to use randomness to design a data structure that can be shipped with the browser, that is as small as possible. The data structure can be queried with a function called `isMalicious`, and has the following three properties:

- **(Small space.)** The data structure uses at most b bits, where b is an integer that is a design parameter you'd like to minimize.
- **(No false negatives.)** If a website x is on the blacklist, then `isMalicious( $x$ )` always returns `True`.
- **(Unlikely false positives.)** Fix a website x that is *not* on the blacklist. Then with probability at least 0.99 over the randomness that you used when choosing the data structure, `isMalicious( $x$ )` returns `False`.

Above, we emphasize that the probability works as follows: the blacklist is first fixed and *will never change*. Fix some website x , either malicious or not, and keep it in mind. Now we use randomness to generate the data structure that will ship with the browser. If x was malicious, then with probability 1, `isMalicious( $x$ )` = `True`. If x was not malicious, then with probability at least 0.99, `isMalicious( $x$ )` = `False`. The only randomness in the problem has to do with the choice of the data structure.

- (a) (1pt) Suppose that we just ship the blacklist directly with the browser (so there is no randomness, and we always correctly classify both malicious and legitimate websites). How many bits does this require? You may assume that a single website (out of all M of them) can be represented using $\log(M)$ bits. **[We are expecting: a single sentence.]**

For parts (b) and (c): Let n be an integer. Let $\mathcal{H}$ be a collection of functions $h : \{x : x \text{ is a website}\} \rightarrow \{1, \dots, n\}$ that map websites to one of n buckets, and suppose that $\mathcal{H}$ is a universal hash family. Notice that the size of the domain of h is M . Suppose that, as with the universal hash family we saw in class, the description of an element $h \in \mathcal{H}$ requires $d = O(\log(M))$ bits.

- (b) (3pts) Below, we describe one way to randomly generate a data structure, using the universal hash family $\mathcal{H}$. The pseudocode to construct the data structure is:

```

Choose h in H at random.
Initialize an array T that stores n bits.
Initially, all entries of T are 0.
for x in the black list:
    T[h(x)] = 1.
end for

```

Then we ship T and a description of h with the browser. In order to query the data set, we define


```

function isMalicious(x):
    if T[h(x)] = 1:
        return True
    else:
        return False
    end if
end function

```

Suppose that $n = 100w$. Show that the three requirements above are met, with $b = d + 100w$.

[We are expecting: a proof that all three properties hold: small space, no false negatives, unlikely false positives.]

- (c) (3 pts) Now consider the following variant on the data structure from the previous part. Let $\mathcal{H}$, d and n be as above. Instead of one array T , we will initialize 10 arrays $T_1, \dots, T_{10}$, as follows:

```

Choose h_1, ..., h_10 independently and uniformly random in H.
Initialize 10 arrays T_1, ..., T_10 that store n bits each.
Initially, for all i=1, ..., 10, all entries of T_i are 0.
for x in the black list:
    for i = 1, ..., 10:
        T_i[h_i(x)] = 1.
    end for
end for

```

We ship $T_1, \dots, T_{10}$, and a description of $h_1, \dots, h_{10}$, with the browser. Now, in order to ask if x is on the black list, we define:

```

function isMalicious(x):
    if T_i[h_i(x)] = 1 for all i = 1, ..., 10:
        return True
    else:
        return False
    end if
end function

```

Suppose that $n = 2w$. Show that all three requirements are met, with $b = 10d + 20w$.

Notice that when w is much bigger than d , this is better than part (b)!

[We are expecting: A proof that all three properties hold. You may reference your proof in part (b) for part (c).]

SOLUTION:

- (a) This requires $b = w \log(M)$ bits. $\log(M)$ bits for each of w sites on the blacklist.
- (b) Suppose that we implement the pseudocode with $n \geq 100w$, as in the problem statement. We verify the three properties.

- We first verify that, no matter what n is, the probability of labeling a malicious website as malicious is 1. To see this, observe that if x is a malicious website, then $T[h(x)] = 1$ because we inserted it when initializing T . But then by definition `isMalicious(x)` returns `True`.
- Next, we consider the probability of accidentally identifying a valid website as malicious: we wish to show that this is less than 0.99. Fix a valid website x . Because of the universal hash family property, for any other website y , the probability that $h(y) = h(x)$ is at most $1/n$. By the union bound over all w malicious y ,

$$P[\text{there is some } y \text{ so that } h(y) = h(x)] \leq \frac{w}{n}.$$

Thus, the probability that this number is less than 1 (aka, zero, since it must be an integer) is at least

$$P[\text{no malicious } y \text{ have } h(x) = h(y)] \geq 1 - \frac{w}{n} = 1 - \frac{w}{100w} = 0.99.$$

But if it's the case the no malicious y have $h(x) = h(y)$, then $T[h(x)] = 0$, and so `isMalicious(x)` correctly reports that x is a legitimate website.

- Finally, we consider the space. We are using $n = 100w$ bits for the table T , and additionally we require d bits to store the hash function h .

(c) Again we verify the three things.

- As in part (a), if x is malicious, then $T_i[h_i(x)] = 1$ for all i , and so it will be identified as malicious with probability 1.
- Next, we fix an x and assume that x is *not* malicious. By the same argument as above, for each i individually we have

$$P[\text{there is any malicious } y \text{ so that } h_i(y) = h_i(x)] \leq \frac{w}{n} = \frac{1}{2}.$$

Now, the probability that x is reported as malicious is the same as the probability that for all $i = 1, \dots, 10$, there is some malicious y_i (which may depend on i) so that $h_i(y_i) = h_i(x)$. That is,

$$P[\text{isMalicious}(x) = \text{True}] = P[\forall i = 1, \dots, 10, \exists y_i, h_i(y_i) = h_i(x)].$$

Because the h_i were chosen independently, we have

$$\begin{aligned} P[\text{isMalicious}(x) = \text{True}] &= P[\forall i = 1, \dots, 10, \exists y_i, h_i(y_i) = h_i(x)] \\ &= (P[\exists y_i, h_i(y_i) = x])^{10} \\ &\leq \left(\frac{1}{2}\right)^{10} \\ &\leq .01. \end{aligned}$$

- Finally, we verify the space. As above, this is the space to store the 10 hash functions, which is $10d$, plus the $20w$ bits for the 10 arrays $T_1, \dots, T_{10}$, each of which have size $n = 2w$.